U.S. Patent Application
Docket #20661-00787USPT

CERTIFICATE OF MAILING BY EXPRESS MAIL

"EXPRESS MAIL" Mailing Label No EL916646410US

Date of Deposit August 16, 2001
I hereby certify that this paper or fee is being deposited with the U S Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and is addressed to Box Patent Application, Commissioner for Patents, Washington, D C 20231

Type or Print Name Keith W Saunders

Signature

Title: ENCRYPTION-BASED SECURITY PROTECTION FOR PROCESSORS

Inventor(s): 1. EDWARD TANGKWAI MA

2. WENDELL L. LITTLE

3. STEPHEN N. GRIDER

4. _____ _. _____

ENCRYPTION-BASED SECURITY PROTECTION FOR PROCESSORS

TECHNICAL FIELD OF THE INVENTION

The principles of the present invention generally relate to processors, and more particularly, by way of example but not limitation, to security microcontrollers with encryption features.

BACKGROUND OF THE INVENTION

Electronic devices are a vital force for creating and perpetuating the engine that drives today's modern economy; concomitantly, electronic devices improve the standard of living of people in our society. Furthermore, they also play an important role in providing entertainment and other enjoyable diversions. A central component of many of these electronic devices are processing units. Processing units may be broadly divided into two categories: (i) processors used as central processing units (CPUs) of (e.g., personal) computers and (ii) embedded processors (a.k.a. microcontrollers, microprocessors, etc.) (e.g., processors operating in cars, microwaves, wireless phones, industrial equipment, televisions, other consumer electronic devices, etc.). Although CPUs of computers garner

the lion's share of reports and stories presented by the popular press, they are only responsible for less than 1% of all processors sold while microcontrollers are actually responsible for greater than 99% of all processors sold. Consequently, significant time and money is also expended for research and development to improve the efficiency, speed, security, feature set, etc. of microcontrollers. These aspects of microcontrollers may be improved, individually or in combination, by improving one or more of the individual aspects of which microcontrollers are composed. Exemplary relevant aspects of microcontrollers include, but are not limited to: processing core, memory, input/output (I/O) capabilities, security provisions, clocks/timers, program flow flexibility, programmability, etc.

Many uses of microcontrollers involve a need for security. The security may be related to the executable code, the data being manipulated, and/or the functioning of the microcontroller. For example, it is important to guard against the possibility of someone monitoring program code and deciphering communication protocols and/or information that are communicated between an automated teller machine (ATM) and the computer at an associated bank. Criminals, hackers, and mischief makers continually attempt to thwart and break existing security measures. Conventional, relatively relaxed and crackable, standards and approaches lead to security deficiencies because it is fairly easy to decipher or to access conventional systems and interfaces, and then control or otherwise jeopardize the

microcontroller's mission. One technique for cracking a microcontroller's security is to monitor information entering the microcontroller and information exiting the microcontroller in order to effectively reverse engineer the program code by revealing what information may be effectuating the execution of a given instruction. It is therefore apparent that newer and stricter security measures are needed to safeguard against information theft, corruption, and/or misuse.

SUMMARY OF THE PRESENT INVENTION

The deficiencies of the prior art are overcome by the methods, systems, and arrangements of the present invention. For example, as heretofore unrecognized, it would be beneficial to utilize block decryption after several reads from external memory. In fact, it would be beneficial if multiple executable instructions were decrypted simultaneously and accessible to a core processor/instruction decoder via, e.g., a decryption buffer, a cache, etc.

In certain embodiment(s), multiple instructions are loaded into a buffer from an external memory. The multiple instructions are decrypted at least substantially simultaneously and then made available to processing core and/or instruction decoder. An instruction desired by the processing core/instruction decoder may be routed directly thereto, or all or a portion of the multiple instructions may be first transferred to a cache. Also in certain embodiment(s), "n" bytes (e.g., an exemplary eight bytes) of encrypted

information may be loaded byte-by-byte (or in chunks of multiple bytes) into an "n"-byte-wide encrypted buffer from an external memory. The "n" encrypted bytes may then be jointly decrypted using a designated encryption/decryption scheme, after which they may be forwarded to an "n"-byte-wide decrypted buffer. A processing core/instruction decoder, possibly in conjunction with a memory management unit (MMU)/memory controller, determines and requests a program instruction address. If the program instruction address hits in an associated instruction cache, then the instruction byte may be retrieved therefrom. If the program instruction address misses the cache but hits in the decrypted buffer, the requested instruction byte may be forwarded immediately to the processing core/instruction decoder while the "n"-bytes of the decrypted buffer are moved into the cache at the appropriate location approximately simultaneously. The buffers and decryptor, possibly in conjunction with the MMU/memory controller, may also be configured for prefetching.

BRIEF DESCRIPTION OF THE DRAWINGS

A more complete understanding of the methods, systems, and arrangements of the present invention may be obtained by reference to the following Detailed Description when taken in conjunction with the accompanying drawings wherein:

FIGURE 1 illustrates an exemplary system block diagram of a microcontroller embedded in an electronic device system;

FIGURES 2A and 2B (collectively FIGURE 2) illustrate an exemplary functional block diagram of the microcontroller of FIGURE 1 with an exemplary decryption feature according to the principles of the present invention;

FIGURE 3 illustrates an exemplary flow chart exemplifying execution of an external program memory decryption and instruction fetch according to the principles of the present invention;

FIGURE 4 illustrates an exemplary functional block diagram of an external memory control and data flow according to the principles of the present invention;

FIGURE 5 illustrates an exemplary functional block diagram of a tag definition and instruction cache according to the principles of the present invention;

FIGURE 6 illustrates an exemplary cache memory structure for an instruction cache according to the principles of the present invention; and

FIGURE 7 illustrates an exemplary flow chart exemplifying execution of an external memory, byte-oriented fetch and decryption according to the principles of the present invention.

DETAILED DESCRIPTION OF THE DRAWINGS

The numerous innovative features of the present application are described with particular reference to the illustrated exemplary embodiments. However, it should be

understood that this class of embodiments provides only a few examples of the many advantageous uses of the innovative teachings herein. In general, statements made in the specification of the present invention do not necessarily delimit any of the various aspects of the claimed invention. Moreover, some statements may apply to some inventive features, but not to others.

Security of systems including processors is problematic in that it can be extremely difficult to protect against program copying, corruption, and/or misuse. Security measures that may be taken include using block encryption of code and/or data, self-destruct inputs, or programmable countermeasures against attacks. External encryption sequencing is a critical aspect of security when fetching external data or instructions stored in an external memory. The information stored in the external memory may be encrypted using any of many encryption approaches and standards, such as, for example, the Data Encryption Standard (DES) algorithm, the triple DES algorithm, the Advanced Encryption Standard (AES) algorithm, etc. While byte-length instructions may be encrypted individually, cracking such encryption is relatively easier because a would-be hacker monitoring the processor may track input instructions and output results on something of a one-to-one basis. Block encryption, on the other hand, may be performed on multiple instructions/bytes simultaneously (e.g., a block of eight instructions composed eight bytes). Employing block encryption, however, may result in processor stalls because multiple encrypted instructions

may be loaded for decryption before a decrypted instruction can be provided to the processing core/instruction decoder. An en/decryption scheme may be designed so as to rely on caching and prefetching (e.g., of blocks of instructions) to reduce or minimize the occurrence of stalls of the microcontroller. Consequently, a well-designed and well-implemented microcontroller in accordance with the principles of the present invention may execute even large loops and jumps without having to stall. In short, improvements to certain security aspects of microcontrollers may be accomplished by modifying the memory, I/O capabilities, security provisions, interrelationships therebetween, etc. of the microcontrollers.

Referring now to Figure 1, there is illustrated an exemplary system block diagram of a microcontroller embedded in an electronic device system according to the principles of the present invention. The microcontroller 100 may control (e.g., perform processing operations for, command actions to be undertaken by, etc.) the entire electronic device system 105, a portion of the electronic device system 105, etc. The microcontroller 100 has application in a variety of systems and/or fields. These systems and/or fields may especially include environments in which high performance and/or low power are important. The systems and/or fields (and hence the nature of the electronic device system 105) in which the microcontroller 100 may be employed include, but are not limited to: telecommunications, industrial controls, hand-held/portable devices, system supervision,

data logging, motor control, etc. The electronic device system 105 may itself correspond to, for example: data switchers and/or routers; subscriber line interface cards; modems; digitally-controlled machining tools such as mills, lathes, drills, etc.; portable radios; cellular telephones; voltmeters, ammeters, and ohmmeters; Personal Digital Assistants (PDAs); televisions; cable or satellite TV set top boxes; camcorders; audio/visual equipment; audio compact disk (CD) systems/players/recorders; digital versatile disk (DVD) systems/players/recorders; financial (especially transaction) equipment such as personal identification number (PIN) pads, point of sale (POS) terminals, etc.; smart cards; etc.

Referring to Figure 2A, there is illustrated an exemplary functional block diagram of the microcontroller of Figure 1 according to the principles of the present invention. The microcontroller 100 is coupled to an external memory 205 through a bus 210. The exemplary microcontroller 100 includes an input/output (I/O) buffer 215, a decryptor 220, a decrypted buffer 225, a cache 230, and a CPU and/or instruction decoder 235 (herein referred to as CPU/instruction decoder 235). It should be noted that the CPU/instruction decoder 235 may alternatively be any subsequent processing or other information utilizing/accepting component or stage. The I/O buffer 215 may be coupled to the external memory 205 (e.g., via the bus 210) and to the decryptor 220, which may be coupled to the decrypted buffer 225. The decrypted buffer 225 may further be coupled to the cache 230, which may be coupled to the CPU/instruction decoder 235 via, for example, an internal bus

240. Advantageously, the decrypted buffer 225 may also be coupled to CPU/instruction decoder 235 (e.g., via the internal bus 240, another bus (not illustrated), etc.) to enable cache bypassing. The microcontroller 100 is able to transfer (e.g., encrypted) information over the bus 210 from the external memory 205 into the I/O buffer 215, where the encrypted information is held until it is forwarded to the decryptor 220. The decryptor 220 is able to decrypt the encrypted information and then forward the decrypted information to the decrypted buffer 225. The decrypted information may be loaded from the decrypted buffer 225 into the cache 230, where it may be accessed by the CPU/instruction decoder 235 for further processing.

It should be noted that in certain embodiment(s) the decrypted information, or at least part thereof, may be forwarded directly from the decrypted buffer 225 to the CPU/instruction decoder 235, thus bypassing the cache 230 for at least that decrypted information. In such embodiment(s), the decrypted information may optionally be substantially simultaneously (e.g., within predictable time delays/lags of circuit elements) loaded into the cache 230. Also, it should be understood that in certain embodiment(s) the microcontroller 100 of Figures 2A and 2B may be implemented using alternative arrangement(s). For example, the buffer preceding the decryptor 220 (i.e., the I/O buffer 215 in Figures 2A and 2B) need not necessarily have direct access to the bus 210, which is illustrated as being external to the microcontroller 100, for there may be one or more

intervening buffer(s) between the I/O buffer 215 and an external bus. Similarly, and as another example, the decrypted buffer 225 need not be separate (or even be an additional buffer) from the decryptor 220, for the decryptor 220 can alternatively hold the decrypted information until it may be forwarded.

Referring to Figure 2B, there is illustrated an exemplary decryption feature according to the principles of the present invention. In operation, in certain exemplary embodiment(s), the I/O buffer 215 receives information in the form of multiple instructions (e.g., $instruction_n$ to $instruction_m$ for a total of "m-n+1" instructions) from the external memory 205 (not explicitly shown in Figure 2B). The I/O buffer 215 may, for example, receive these instructions consecutively (instruction-by-instruction), in sets (two or more instructions substantially simultaneously), together (all of the instructions substantially simultaneously), etc. All of the encrypted instructions in the I/O buffer 215 may be substantially simultaneously forwarded to the decryptor 220. The decryptor 220 may then use a decryption algorithm (e.g., DES, triple DES, AES, etc.) to perform a decryption operation on all of the encrypted instructions forwarded thereto. Advantageously, the overall decryption operation is therefore performed essentially simultaneously on at least two instructions. The decrypted instructions may thereafter be forwarded from the decryptor 220 to the decrypted buffer 225 for holding. If one or more of the decrypted instructions has not already been requested by the CPU/instruction decoder 235 (of Figure

2A), the decrypted instructions may remain in the decrypted buffer 225 until at least one of the decrypted instruction therein is requested by the CPU/instruction decoder 235 for execution. Once requested, the requested decrypted instruction may be forwarded directly to the CPU/instruction decoder 235 for execution while all instructions from the decrypted buffer 225 are substantially simultaneously moved to the cache 230. Alternatively, the CPU/instruction decoder 235 may wait until the decrypted instructions are stored in the cache 230 and retrieve the requested decrypted instruction therefrom.

Referring to Figure 3, there is illustrated an exemplary flow chart exemplifying execution of an external program memory decryption and instruction fetch according to the principles of the present invention. At step 305, multiple encrypted instructions are received into a buffer of a microcontroller from an external memory. A selected encryption process (and related encryption algorithm) was previously performed on these instructions as or before, for example, they were stored at the external memory. At step 310, the multiple encrypted instructions are decrypted (e.g., substantially simultaneously) using a selected decryption algorithm (e.g., the decrypted algorithm corresponding to the previously-selected encryption algorithm). At step 315, at least one of the decrypted instructions is transferred to a processing area of the microcontroller. The processing area of the microcontroller may correspond to, for example, a central processing unit (CPU), an instruction decoder, an instruction execution unit, etc. The steps may be repeated to implement on-demand

retrieval, predictive pre-fetching, etc. from the external memory so as to feed the processing area of the microcontroller a continuing stream of instructions with minimal or at least reduced delays.

Referring to Figure 4, there is illustrated an exemplary functional block diagram of an external memory control and data flow according to the principles of the present invention. The microcontroller 100, according to the principles of the present invention, implements block encryption for program memory with a block size of "n" bytes, where "n=8" in the exemplary illustrated embodiment. A CPU/instruction decoder (not explicitly illustrated in Figure 4) primarily executes instructions from either the internal program memory space or the cache 230. Program code is fetched from an external memory (not explicitly illustrated in Figure 4) (e.g., from an I/O buffer 215') to the program decryptor 400 one byte at a time (or multiple bytes at a time if the external bus is wider than one byte) until an 8-byte block has been loaded, which may then be written into the cache 230 after a decryption process. The cache may be realized using, for example, a 2-way set associative instruction cache with 1 K bytes of memory space. The tags 410 for each block of each way include, for example, the 13 high order address bits and two status bits for validation and replacement. Program counter (PC) addresses are compared to the tags 410 to determine one of the following instances:

(1) A hit occurs if there is a match and the corresponding tag is valid; or

(2) A miss occurs if there is no match or if there is a match but the tag is invalid. A hit enables the corresponding instruction or operand byte to be used for execution. A miss results in a stall of the program execution, waiting for the needed block to be loaded and decrypted. Address generation logic 415 provides the external addresses, for example, to program memory for external instruction fetch. The internal address bus 240A provides the addresses for accessing the cache 230 on reads and writes.

External data accesses are transferred through the data encryptor 420. The block size for external data memory encryption may be an exemplary one byte, but multiple-byte blocks may alternatively be implemented. If the corresponding encryption enable bit of a particular data memory chip is cleared to a logic 0, for example, the data may be transferred directly to/from the accumulator and the data memory chip. Thus, no encryption/decryption need be involved. The memory controller 405 may be responsible for coordinating the internal address bus 240A and the internal data bus 240D activities, the cache/tag access and update, program block decryption, and external data/program memory accesses.

Program decryption, on the other hand, may be effectuated on blocks of 64 bits through the program decryptor 400. This program decryptor 400 may be based, for example, on the full 16-round DES algorithm and may also be capable of supporting single or triple DES operations. The microcontroller 100 may, for example, be designed to decrypt a 64-bit block in five machine cycles for single DES operations and to decrypt a 64-

bit block in 7 machine cycles for triple DES operations. The program decryptor 400 may include the following exemplary units and/or aspects:

(1) A 64-bit encrypted buffer;

(i) Capable of interfacing a byte-wide data bus (e.g., an embodiment of the bus 210 of Figures 2A and 2B) directly or via the Data I/O buffer 215';

(ii) Data transfer between the encrypted buffer and the byte-wide data bus is 8-bits wide; and

(iii) Capable of loading encrypted data to a decryption unit in 64-bit-wide blocks; and

(2) A 64-bit block decryption unit;

(i) Capable of unloading decrypted data to the cache 230 in 64-bit-wide blocks (e.g., via a decrypted buffer);

(ii) Unloading of data to internal data bus 240D is 8-bits wide; and

(iii) Supports direct execution by a CPU/instruction decoder in ROM loader mode.

The DES algorithm, which may be employed in accordance with the present invention, is based on the principle of building up a sequence of simple operations to form an overall complex operation, each round of operation providing very little security if separated. The basic operation of each round consists of some permutations and

substitutions, determined by a subset of the key bits, performed on one half of the data. In this operation, one-half may be encrypted and the other half may continually pass through unchanged, providing invertible paths for decryption. The operation can be performed using the following equations:

Given an initial input data M divided into left and right halves (L_0, R_0), M is transformed after the i^{th} round into $M_i = (L_i, R_i)$ defined by

$$L_i = R_{i-1} \quad (1)$$

$$R_i = L_{i-1} + f(R_{i-1}, K_i) \quad (2)$$

where K_i is the encryption subkey. This is easily invertible if the key is known since, given (L_i, R_i), it may recover (L_{i-1}, R_{i-1}) by

$$L_{i-1} = R_i + f(R_{i-1}, K_i) = R_i + f(L_i, K_i) \quad (3)$$

$$R_{i-1} = L_i \quad (4)$$

The f function itself need not be invertible, any desired f function may be used. However, the f function used in the DES is designed to provide a high level of security by a specially chosen base 2 value in the so called S-boxes.

The 16 subkeys can be generated from the 56 key bits (originally generated by a random number generator (An exemplary random number generator that may be employed along with the principles of the present invention is described in U.S. Nonprovisional Application for Patent Serial No. 09/879,686, filed on June 12, 2001, and entitled "IMPROVED RANDOM NUMBER GENERATOR". U.S. Nonprovisional Application for Patent Serial No. 09/879,686 is hereby incorporated by reference in its entirety herein.)) by first applying an initial 56-bit permutation to the original 56 key bits and then using the shifting sequence (1,1,2,2,2,2,2,2,1,2,2,2). A permutation of length 48 is applied after each

shift to pick out the subkeys.

A predefined permutation is used to permute the 64 data bits before dividing into two halves. The f function is performed on one half of the input data during a round of process. These data bits are mapped into a 8x6 diagram as the following:

$$n_{31} \mid n_0\ n_1\ n2\ n3 \mid n_4 \qquad (5)$$

$$n3 \mid n_4\ n_5\ n_6\ n_7 \mid n_8$$

$$: \mid \quad : \quad \mid :$$

$$n_{27} \mid n_{28}\ n_{29}\ n_{30}\ n_{31} \mid n_0$$

Modulo 2 addition of the above 48 numbers with the subkey bits provide the reference for the result of the f function from the 4x 16 S-box. The 8 S-boxes can be referenced from the Data Encryption Standard (DES). This round of encryption is completed by modulo 2 addition of the result base 2 numbers with the other half data bits after a predefined permutation.

This process is repeated through all 16 rounds with a different set of subkeys. A final permutation then takes place which is the inverse of the initial permutation.

The security of a system having a processor and a memory, for example in accordance with certain embodiment(s) of the present invention, can be further improved if the encryption/decryption keys are modified by, for example, the address of the block (or other level of addressable granularity such as the byte, word, page, etc.) that is being encrypted/decrypted. Such a modification effectively creates a different key for each block (or other level of granularity) that is being encrypted/decrypted. Advantageously, if the key

is dependent on the block address, no two (2) blocks can be swapped with each other or otherwise moved. When blocks can be swapped or otherwise moved, then another avenue of attack is available to a would-be hacker because the would-be hacker is able to change the order of execution of the system. Limiting or blocking this avenue of attack using the modification scheme described in this paragraph can be accomplished in many ways. By way of example but not limitation, this modification may be effectuated by (i) having the relevant address "xor"ed with the key of (or with more than one key if, e.g., the triple DES is selected as) the encryptor/decryptor algorithm, (ii) having the relevant address added to the key or keys, (iii) having the relevant address affect the key(s) in a non-linear method such as through a table lookup operation, (iv) having one or more shifts of the relevant address during another action such as that of (i) or (ii), some combination of these, etc.

While the modification described in the above paragraph can cause code under attack to become garbled and useless, the following modification described in this and the succeeding paragraph can recognize that code is or has become garbage and, optionally, take an action such as destroying the code, destroying or erasing one or more keys, destroying part(s) or all of the chip, etc. It should be noted that either modification may be used singularly or together in conjunction with certain embodiment(s) in accordance with the present invention. For this modification, an integrity check can be added to the system so as to ensure that the integrity of the code has not been compromised by an attacker. Such

an integrity check may be accomplished in many ways. By way of example but not limitation, this modification may be effectuated by fetching a checksum byte or bytes after a block of code is fetched. This fetched checksum may be associated (e.g., by proximity, addressability, a correspondence table or algorithm, etc.) with the block of code previously fetched. It should be noted that the addressing order/location, as well as the fetching order, of the fetched block and the fetched checksum may be changed. It is possible, for example, to use different busses and/or different RAMs to store the encrypted code and the checksum(s), but in a presently preferred embodiment, each is stored in different section(s) of the same RAM (but may alternatively be stored in the same section).

After the block of encrypted code is fetched and decrypted, it may be latched into a checksum calculation circuit (or have a checksum operation performed in the same "circuit" as that of the decryption). A calculated checksum of the decrypted code is calculated by the checksum circuit/operation. The calculated checksum may be compared with the fetched checksum. If they differ, then the block of code may be considered to have failed the integrity check. The calculated checksum may be calculated in many different ways. By way of example but not limitation, the checksum (i) may be an "xor" of the fetched block of information, (ii) may be a summation of the fetched block of information, (iii) may be a CRC of the fetched block of information, (iv) may be some combination thereof, etc. If the fetched block fails the checksum comparison, then various exemplary

actions may be taken by the system/chip. By way of example but not limitation, actions that may be taken responsive to a failed checksum integrity check include: (i) a destructive reset that may, for example, result in the clearing of internal key information and/or internal RAMs, (ii) an evasive action sequence may be started, (iii) an interrupt may allow the system program/chip to take action, (iv) some combination thereof, etc.

The microcontroller 100 of Figure 4, according to certain embodiment(s)of the present invention, supports single DES or triple DES operations, and uses two 56-bit keys for its program encryption/decryption. It should be understood that the same key, even if termed a decryption key or an encryption key, may be used in both the encryption and the decryption process. However, the program decryptor 400 may be defaulted to perform single DES operations. The triple DES function is then enabled by setting/clearing a bit in a special function register (SFR). The time required to compete a triple DES operation is longer than that for completing a single DES operation, so there is a latency/security tradeoff in the selection between the different standards. It should be noted that other standards may be utilized instead of a DES standard.

In certain embodiment(s), the program decryptor 400 may be accessed by loaders via specific program block decryption registers in the SFR. The loaders can select either an encryption or a decryption operation by setting/clearing the relevant bits when using the program decryptor 400. The data encryptor 420 may perform byte encryption and

decryption on data transferred to and from external memory (not explicitly illustrated in Figure 4) using one of four data transfer instructions. Eight bytes representing the 64-bit key may be written to a data encryption key (DEK) location in the SFRs before using the data encryptor 420. This can be accomplished, for example, during a ROM boot loader mode. Loading the key may be effectuated by implementing eight consecutive SFR writes to the DEK register. The data encryptor 420 is a byte encryption/decryption logic unit. Data transfer instructions cause the data encryptor 420 to perform a required operation on a selected data byte. Encryption/decryption operations require an additional machine cycle to complete a data transfer instruction. However, any required memory access stretching may be accommodated internally.

Referring to Figure 5, there is illustrated an exemplary functional block diagram of a tag definition and instruction cache according to the principles of the present invention. The microcontroller 100, according to certain embodiment(s) of the present invention, may incorporate a 2-way set associative cache to enhance system performance. A cache way in the exemplary embodiment of Figure 5 may be viewed as 64 blocks of 8 byte contiguous memory spaces. Blocks with the same index address can be stored in either way 0 230a or way 1 230b at their corresponding index locations. The cache 230 may be a lKx8 RAM that is logically divided into 2x5l2x8 program spaces to support the two way cache structure. One 64x28 SRAM (logically divided into 2x64xl4) may be used for tags 410 to identify the

blocks and their validation. Each tag 410a and 410b contains the 13 high order address bits 505 corresponding to the instruction data byte, a valid bit 510, and a status bit 520. The 13 high order address bits 505 represent the tag 410 itself, and the status bit 520 is used to implement a least recently used (LRU) algorithm, for example, for possible block replacement. Each memory block is validated by its corresponding valid bit 510. The internal program address (e.g., the program counter), which can be made known to the memory controller 405, is an exemplary 22 bits. These 22 bits are composed of the thirteen (13) highest order [21:9] bits 525, the next six (6) bits [8:3] that represent the block index address 515, and the lowest order two (2) bits [2:0] that represent the byte offset 530. It should be noted that the number of bits in the internal address may be greater than 22, such as, for example, 24 bits, 32 bits, 64 bits, etc. It should be understood that the cache configuration (e.g., arrangement, size, ways, replacement scheme, etc.) can vary without departing from the principles of the present invention. Likewise, the number of address bits and the division thereof between block index, block offset, etc. may vary without departing from the principles of the present invention.

Tags 410 may be made invalid on reset, and the, e.g., byte-wide internal address bus 240A may be initialized to the first external program memory block with byte offset 0h. After a reset, the memory controller 405 may begin fetching program code from the external memory. Eight consecutive external addresses are generated by the address generator 415

(illustrated in Figure 4) to fetch eight bytes of code to the encrypted buffer 400EB. The 64-bit block of encrypted code is then decrypted with the decryptor 400D, with the results being placed in the decrypted buffer 400DB. (It should be understood that the decryptor 400D may also optionally include encrypting functions for use such as, for example, during a boot up mode in which the instructions are being encrypted and written to external memory. This is also true in general where applicable, and vice versa.) The decrypted block of code may then be loaded into the cache 230 when/if requested. Instruction data byte blocks written from the program decryptor 400 to the cache 230 are 64-bits wide. The corresponding tag 410a or 410b is updated and validated by setting/clearing the valid bit 510 (The bit is set or cleared depending on the logic value selected for indicating valid or invalid.). The address generator 415 continues to (pre-)fetch the next block of code when a block is transferred to the decryptor 400D. The memory controller 405 attempts to stay at least a block ahead of the address of execution, to thus have the next sequential block of decrypted code ready for execution when (possibly and hopefully) requested. Therefore, when a block with index address X is loaded into the cache 230, the block with index address X+1 is next loaded into the decryptor 400D (if done loading into the encryption buffer 400EB), and external (pre-)fetching begins on the block with index address X+2 into the encryption buffer 400EB. In this manner, stalls of execution for straight-line code, once the instruction fetch pipeline is full, are significantly reduced, or even eliminated.

The Least Recently Used (LRU) bit 520 of each tag 410 is used by the memory controller 405 to determine which block is replaced upon a write to the cache 230. An access to a byte in a block clears/sets that block's LRU bit 520 and sets/clears the corresponding block's LRU bit 520 in the other cache way 230a or 230b at the same index address. An exemplary cache block replacement policy is as follows:

(1) Replace the invalid block. If both tags 410 are invalid, replace the block in Way 0 230a; and

(2) If there are no invalid blocks, replace the LRU block.

Replacing a valid block causes the necessity of reloading that particular block if accessed again. A newly written block is set to be valid and to have the LRU bit(s) updated.

The MMU is responsible for managing and operating the cache 230, the program and data encryptors 400 and 420, and the accessing of external memory 205. The MMU may include the cache control, the program decryptor 400, and the external address generator 415. The memory controller 405 may provide the following exemplary control functions:

(1) Read/write access to the cache 230;

(2) Read/write access to the external memory 205;

(3) Tag 410 accessing and updating;

(4) Byte-wide bus sequencing and timing control;

(5) Address generation for external memory 205;

(6) Loading and unloading of the program decryptor 400; and

(7) Hit and miss (stall) signal generation.

The memory controller 405 controls the cache 230 read/write access by monitoring the PC address activities and comparing the PC address with the tags 410. A hit enables a cache read, a miss causes a stall of the CPU clocks while the memory controller 405 fetches the needed block of code from the external memory 205 (if not already somewhere in the pipeline such as in the program decryptor 400). The requested code block is placed into one of the two appropriate index locations according to the active replacement policy. The corresponding tag 410a or 410b is also updated while normal program execution resumes.

The internal address is latched by the address generator 415 during a data transfer instruction or during a complete miss (block is also not in pipeline). Otherwise, the address generator 415 simply fetches the next sequential block after passing a block to the decryptor 400D. The internal address provides the address to the cache 230a or 230b and tags 410a and 410b. As illustrated in Figure 5, the cache 230a and 230b and tags 410a and 410b are also addressed by the block index bits [8:3] 515. The available cache output (64 bits) based on the tag and block index is decoded further with the byte offset [2:0] 530 in order to retrieve the desired instruction byte from the eight instruction bytes in the block. The block index address represents and determines the two predetermined locations (if a two-way cache) that a block of code with an identical block index value can be found in the cache

230; similarly, the block index address dictates which two tags from the tags 410a and 410b need to be output for comparison to the 13 high order bits of the current PC address.

With respect to specific exemplary implementation(s), external memory 205 may be initialized by a ROM boot loader built into the microcontroller 100, according to certain embodiment(s) of the present invention. When the loader is invoked, the external program memory space appears as a data memory to the ROM and can therefore be initialized. Invoking the loader causes the loader to generate new encryption keys (e.g., from a random number generator), thus invalidating all information external to the part. Invocation of the loader also invalidates the tags 410 and erases the cache 230. The program loading process may be implemented by the following exemplary procedures: the program is read through a serial port, encrypted, and written to the external memory space via data transfer instructions. The encryption process is done in blocks of an exemplary 8 bytes. Eight consecutive bytes are loaded to the program decryption block through an SFR interface, encrypted, then read out. It should be noted that the SFR interface, in certain embodiment(s), may only be available while executing from the loader in ROM mode or a user loader mode. The encrypted code is then written by the loader to the external memory 205 through the byte-wide data bus (e.g., a byte-wide embodiment of bus 210). The byte-wide address bus is driven by the address generator 415 on a data transfer instruction. Eight consecutive memory writes are required to store one block of encrypted code. Data-

transfer-instruction-type information transferred between the accumulator and the external memory 205 may travel via a bus 425 if the information is intended for program space. The data information may or may not be encrypted (depending on the value of encryption select bits). The address generator 415 latches the data transfer instruction address, and the data encryptor 420 drives or reads the byte-wise data bus during the data transfer instruction. With respect to these exemplary implementation(s), data encryption is thus byte-wise and real-time, which contrasts with program encryption, which is a block encryption taking multiple cycles.

Referring to Figure 6, there is illustrated an exemplary cache memory structure for an instruction cache according to the principles of the present invention. A lKx8 RAM, which may be used as the cache 230, is logically configured to 2x5l2x8 to support a 2-way cache structure. Illustrated in Figure 6 is an exemplary RAM configuration with an accessing/addressing scheme. This RAM may have the following exemplary features:

(1) Write data in a 64-bit block from the program decryptor 400;
(2) Read data in a 64-bit block output decoded further to 8-bit data for internal bus;
(3) Addressed by a way select and PC index address bits (e.g., bits [8:3]); and
(4) 2x5l2x8 logically.

The RAM may be implemented as a lKbyte SRAM to save silicon space. The read and write lines and way select line are provided by the memory controller 405, depending on the

tag output of the current index. The index address is provided by the program counter (PC).

Similar RAM structure can be used for the cache tags with the following exemplary features:

(1) Addressing by the block index address (PC [8:3]) 515;

(2) Write operation inputs from two sources;

(i) Cache tag: 13-bit high order address bits [21:9] 525; and

(ii) Two status bits from the memory controller 405;

(3) Read operation outputs to two destinations;

(i) 13-bit tag 505 to the comparator; and

(ii) Two status bits to the memory controller 405;

(4) One 64x28 RAM is needed to support accessing both tags 410a and 410b simultaneously;

(5) The two status bits require individual accessing capability; and

(6) Fast access time.

There need be no timing difference between accessing the cache 230 and any internal SRAM used for storing code.

It should be noted that the hit signal is an important path for a cache read. A cache read includes propagation delays caused by the tag RAM read, tag value comparison, and cache read access, to be performed all in one machine cycle. The hit/miss signals are therefore generated quickly (e.g., in less than two oscillator clocks). If a read miss occurs,

the CPU/instruction decoder or equivalent stalls until the targeted code block is ready to be placed into the cache 230. There are many levels of stall penalties for a single DES read miss.

These include the following exemplary latencies for single DES operations:

(1) Cache miss, hit on decrypted buffer (finished): 0 machine cycles;
(2) Cache miss, hit on decrypted buffer (in progress): 1-4 machine cycles;
(3) Cache miss, hit on encrypted buffer (done loading): 5 machine cycles;
(4) Cache miss, hit on encrypted buffer (in progress): 6-13 machine cycles; and
(5) Cache miss, complete miss: 14 machine cycles.

The stall penalties for a triple DES miss include the following exemplary latencies:

(1) Cache miss, hit on decrypted buffer (finished): 0 machine cycles;
(2) Cache miss, hit on decrypted buffer (in progress): 1-6 machine cycles;
(3) Cache miss, hit on encrypted buffer (done loading): 7 machine cycles;
(4) Cache miss, hit on encrypted buffer (in progress): 8-15 machine cycles; and
(5) Cache miss, complete miss: 16 machine cycles.

Hits on the finished decrypted buffer 400DB do not cause a miss. The CPU/instruction decoder or equivalent can execute from the decrypted buffer 400DB directly, while the corresponding block in cache 230a or 230b is replaced with the buffer instruction data.

Referring now to Figure 7, there is illustrated an exemplary flow chart exemplifying execution of an external memory, byte-oriented fetch and decryption according to the principles of the present invention. Code stored in an external (e.g., program, instruction code, etc.) memory is encrypted (e.g., using the DES algorithm) in blocks of eight bytes; thus, at step 705, external code fetches involve 8 consecutive byte-wise (i.e., totaling a 64-bit block) reads from the external program memory to an encrypted buffer. At step 710, the 64-bit block of instruction data is decrypted using a decryption algorithm, such as the DES algorithm, which may be selected so as to correspond to the encryption algorithm previously used on the instruction data. The eight decrypted bytes are moved (e.g., in a block) to a decrypted buffer at step 715. If at least one instruction from the block is (or has been previously) requested, then, at step 720a, the decrypted block of data is loaded into a cache. It should be noted that data written from the decrypted buffer to the cache is 64-bits wide in this exemplary embodiment, but other bit widths may be supplanted therefore. At step 720b, the tag corresponding to the location of the cache at which the block was loaded is updated and validated by setting/clearing a valid bit. At step 720c, the requested byte of (instruction) data is transferred to a data bus for forwarding to be used for program execution. It should be noted that steps 720a, 720b, and 720c may be performed in a different order, substantially simultaneously, etc. At step 725, processing continues (e.g., a program counter (PC) is updated). For example, as indicated by the dashed arrow, an

address generator continues to (pre-)fetch the next block of code when a block is transferred to a decryption unit (e.g., by returning to steps 705, 710, et seq.). It should be noted, as also indicated by the dashed arrow, that program execution and code fetching can be performed simultaneously, except in the case of stalls when a program execution unit waits until the required code is fetched, decrypted, and then provided thereto.

Instruction fetching from the external program memory (from the perspective of the program execution unit) is actually a read from the cache. To that end, the cache is accessed, or at least checked (along with the address of the block in the decrypted buffer), prior to external code fetching. It should be noted that internal program memory may also be present and accessed. To check the cache, at steps 730a and 730b, 13 high order PC address bits are compared with values of the cache tags that are addressable by a block index (e.g., PC bits [8:3]). It should be noted that other total numbers as well as divisions of the address bits may alternatively be employed and that steps 730a and 730b may be performed in a different order, substantially simultaneously, etc. If there is a match (at step 730b) and the tag is valid (at step 730a), then, at step 735, a memory controller/MMU generates a hit signal and allows a read access to that particular cache location at step 740. If, on the other hand, a cache miss occurs (as determined at steps 730a and 730b), but the PC address matches the block that is currently in the decrypted buffer (and the decryption process is completed) at step 745, the corresponding instruction byte is transferred directly

to the data bus (at step 720c) for the program execution unit, the cache and tags are updated (at steps 720a and 720b, respectively), and execution continues without a stall (at step 725).

A stall occurs when there is no tag match or the tag is matched but invalid and when the data is not available in the decrypted buffer. Such a miss temporarily stalls program execution, and if the desired instruction(s) are not already in the decryption unit or currently being loaded thereto from the encrypted buffer as determined at step 750, the current PC address is used to fetch the desired code from the external program memory (at steps 705, 710, et. seq.). If the PC address does match that of the encrypted instructions, then flow can continue at the decryption or forwarding-to-the-decryption unit stages (at steps 710, 715, et seq.). It should be understood that the PC address is not necessarily the first byte of the block address; however, the memory controller/MMU may set the byte offset bits to 0h for block alignment in the buffers, cache, etc. Program execution may be resumed after the targeted code block reaches the point (at step 725) where the decryption process is finished and the opcode/operand byte may be fed to the program execution unit directly from the decrypted buffer (at step 720c) (e.g., while the cache and tags are updated).

To minimize or reduce the latency associated with external code fetching, processors may be designed with the assumption that program execution is frequently sequential and often follows a rule of locality (e.g., the 90/10 rule). The memory controller/MMU may

(pre-)fetch the next consecutive code block, decrypts it, and has it ready in the decrypted buffer. The memory controller/MMU is typically responsible for controlling these activities, and it is responsible for checking possible matches between the current PC address and the blocks in the encrypted buffer, decryption unit, and decrypted buffer. For example, if the current PC address matches the block in the decrypted buffer (as determinable at step 745), the requested byte of instruction data in the decrypted buffer is used for program execution (at step 725 via step 720c) while the related block is written to the cache (at step 720a). If, on the other hand, the current PC address matches the block in the encrypted buffer (or optionally the decryption unit) (at step 750), the encrypted instruction data in the encrypted buffer is transferred to the decryption unit and decrypted (at step 710). A match to either buffer causes the instruction data therein to be placed in the cache at the appropriate address-based index. In the case of a match to the decrypted buffer, the decrypted instruction data can be placed into the cache without delay; otherwise, a stall occurs until the block having the requested instruction byte is decrypted and forwarded to the decrypted buffer, whereafter the decrypted instruction byte may be driven for execution and the cache and tags updated. As described above, upon a miss of the cache, the requested instruction byte may be driven directly from the decrypted buffer while also being substantially simultaneously written to the cache. Any other instruction bytes in the same block that are subsequently requested are located in and may read from the cache, until the block is

replaced or rendered invalid.

Although embodiment(s) of the methods, systems, and arrangements of the present invention have been illustrated in the accompanying Drawings and described in the foregoing Detailed Description, it will be understood that the present invention is not limited to the embodiment(s) disclosed, but is capable of numerous rearrangements, modifications, and substitutions without departing from the spirit and scope of the present invention as set forth and defined by the following claims.